

Daisen: A Framework for Visualizing Detailed GPU Execution

Yifan Sun¹, Yixuan Zhang², Ali Mosallaei³, Michael D. Shah⁴, Cody Dunne⁴, and David Kaeli⁴

¹ William & Mary, ² Georgia Institute of Technology, ³ Lexington High School, ⁴ Northeastern University

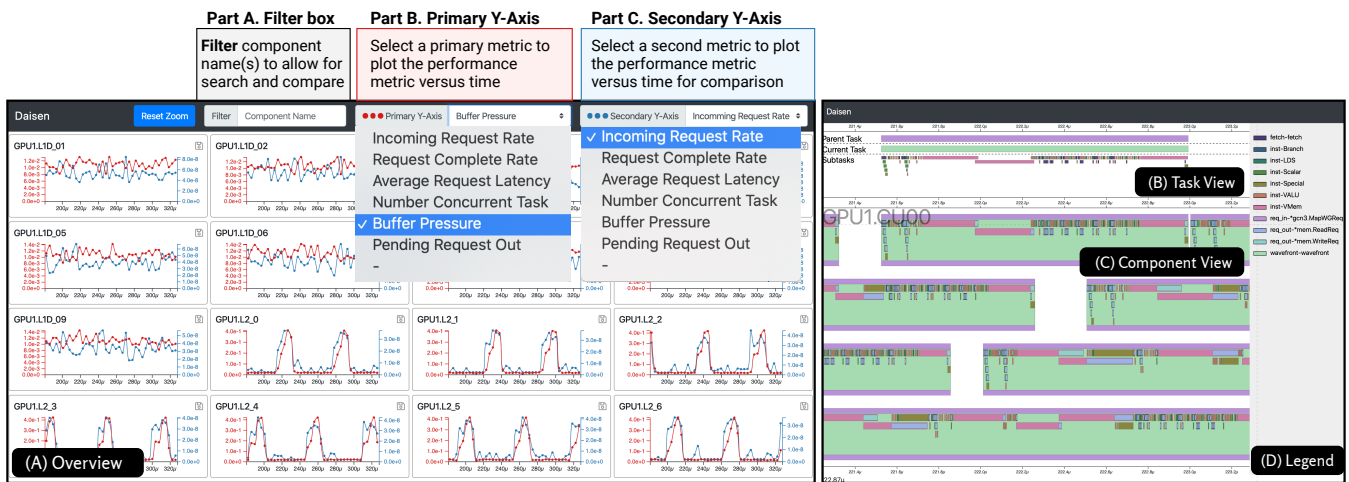


Figure 1: Daisen provides a web-based interactive visualization tool that enables the examination of GPU execution traces. (A) The Overview Panel shows key performance metrics against time of all hardware components using small multiples, (B) the Task View demonstrates the hierarchical relationships between tasks, (C) the Component View displays all the tasks executed in a hardware component (after selecting one component of interest from the Overview), and (D) the legend bar shows all the tasks involved in the Component View and the Task View with color encoded. Note that the labels ABCD do not appear on the visualization interface.

Abstract

Graphics Processing Units (GPUs) have been widely used to accelerate artificial intelligence, physics simulation, medical imaging, and information visualization applications. To improve GPU performance, GPU hardware designers need to identify performance issues by inspecting a huge amount of simulator-generated traces. Visualizing the execution traces can reduce the cognitive burden of users and facilitate making sense of behaviors of GPU hardware components. In this paper, we first formalize the process of GPU performance analysis and characterize the design requirements of visualizing execution traces based on a survey study and interviews with GPU hardware designers. We contribute data and task abstraction for GPU performance analysis. Based on our task analysis, we propose Daisen, a framework that supports data collection from GPU simulators and provides visualization of the simulator-generated GPU execution traces. Daisen features a data abstraction and trace format that can record simulator-generated GPU execution traces. Daisen also includes a web-based visualization tool that helps GPU hardware designers examine GPU execution traces, identify performance bottlenecks, and verify performance improvement. Our qualitative evaluation with GPU hardware designers demonstrates that the design of Daisen reflects the typical workflow of GPU hardware designers. Using Daisen, participants were able to effectively identify potential performance bottlenecks and opportunities for performance improvement. The open-sourced implementation of Daisen can be found at gitlab.com/akita/vis. Supplemental materials including a demo video, survey questions, evaluation study guide, and post-study evaluation survey are available at osf.io/j5ghq.

CCS Concepts

• Computer systems organization → Single instruction, multiple data; • Human-centered computing → Information visualization;

1. Introduction

Graphics Processing Units (GPUs) were originally designed to accelerate 3D graphics rendering. In the last decade, developers have found additional general-purpose use cases that take advantage of GPU's architectures. Developers have been using GPUs to accelerate a wide range of applications that require high performance, including large-scale physics simulation [NHKM14], medical imaging [YTZ*08], data visualization [ME09], artificial intelligence [OJ04], and blockchain hashing [MM18]. A GPU is capable of performing thousands of calculations in parallel. Empirical studies have demonstrated that GPU's massive parallel processing capabilities can speed up algorithms by 5-70X as compared to CPU implementations [ANM*12, SASK19].

GPU hardware designers aim to improve the design of a GPU to run applications faster. Prior to implementing a new design on real hardware and sending the design to fabrication, extensive simulations of the proposed design must be carried out. Simulations allow hardware designers to compare the performance of the proposed designs with a baseline over a wide range of benchmarks (i.e., standard and representative applications) [BYF*09, UJM*12, GUK17, SBM*19, KSAR20]. It is challenging to identify and make sense of performance bottlenecks encountered, since this may involve the designer manually inspecting gigabytes (GBs) of simulator-generated GPU-execution traces (i.e., highly detailed records of events that occur during GPU program execution) [ZUSK15]. Given the complexity and the massive scale of the execution trace data, visualizations can help GPU hardware designers inspect execution states, identify potential performance issues, and improve the efficiency of the GPU hardware design process [AFTA10].

Existing performance analysis tools [GKM82] typically assume the underlying execution data is organized using call graphs (a graph representation where the nodes are functions and the links are function calls). While call graphs are a reasonable representation for tracking software execution, they are not suitable for hardware execution. The problem is that a "function" is a software concept and is not used by hardware. Several visualization tools [IBJ*14, RWP*19, XXM18, FMR*17] have been developed to visualize parallel execution. However, these tools mainly focus on visualizing high-performance distributed computing and cannot be directly applied to hardware design. Visualization tools for examining GPU execution traces, such as AerialVision [AFTA10] and M2S-Visual [ZUSK15], only show specific metrics for specific features (e.g., instruction execution, cache access). Additionally, these tools lack a straightforward methodology to compare metrics when evaluating complex hardware tradeoffs.

We introduce Daisen, a framework that helps GPU hardware designers collect and visualize GPU execution traces. We contribute:

- A task analysis and task abstraction of GPU performance analysis. To the best of our knowledge, there is no formal study that examines the design requirements of GPU execution trace visualization to facilitate GPU hardware design. To fill this gap, we conduct a survey and interviews with domain experts to characterize practices and workflows in GPU performance analysis.
- A data abstraction that is designed to capture simulator-generated traces of GPU executions.

- The design and implementation of Daisen, a web-based visualization tool that visualizes massively-parallel GPU execution traces. Our evaluation with domain experts demonstrates that Daisen can help GPU hardware designers identify performance bottlenecks, make informed and evidence-based design decisions, and efficiently debug performance issues.

2. Background on How GPUs Work

GPUs are computing devices that can perform a large number of data-processing operations concurrently. In this section, we briefly describe how GPUs and GPU simulators work, and summarize the challenges of visualizing highly parallel hardware architectures.

GPU programming. GPUs work alongside CPUs in a coordinated effort to process data. A programmer writes a CPU program (i.e., host program) that prepares data and then copies the data over to the GPU to process. After the required data is copied, the host program starts the GPU program (i.e., launching kernels). GPU vendors usually provide programming APIs (e.g., CUDA, OpenCL) to perform memory copy and kernel launching tasks.

A program that runs on a GPU is called a *kernel*. As depicted in Figure 2 (a), a kernel is composed of a large number of work-items that can be executed concurrently on a GPU. A *work-item* is similar to a CPU thread that runs a series of instructions (e.g., add, multiply) to process data. Several work-items (typically 32 or 64, determined by the hardware) can be grouped into a *wavefront*. Wavefronts (typically 1–8) are further grouped into a *work-group*.

Software elements (e.g., kernels, work-groups, wavefronts, work-items) need to be mapped to hardware resources for execution. We illustrate how the kernel is executed on a GPU with Figure 2 (b) and (c). The CPU launches the whole kernel on a GPU for execution. On the GPU side, the *Command Processor* receives the kernel launch request from the CPU and divides the kernel into work-groups. After this, the Command Processor dispatches the work-groups to the *Compute Units*. Compute Units, similar to CPU cores, are responsible for calculating the program output. When work-groups are dispatched to a Compute Unit, the work-groups are broken down into wavefronts. Instructions that make up each wavefront wait to be scheduled by a central scheduler in each Compute Unit. The central scheduler dispatches instructions to the proper execution units (e.g., Branch Unit, SIMD unit), according to the type of each instruction. For a majority of the calculations, the instruction is sent to the *SIMD units* (digital circuits that can process instructions from multiple work-items at the same time).

GPUs can process many work-items concurrently. E.g., the AMD R9 Nano GPU [AMD15] (default to MGPUSim and used in our case study and evaluation) has 64 Compute Units; each Compute Unit has 4 SIMD units, and each SIMD unit can calculate instructions from 16 work-items in one cycle. Thus, the whole GPU can calculate $64 \times 4 \times 16 = 4096$ instructions simultaneously. As the GPU runs at a 1 GHz frequency (1 nanosecond per cycle), it can execute 4 trillion instructions in 1 second with all the SIMD units. Given this volume of instructions that a GPU runs, it is essential to support visualizing this large dataset while still providing the capability of interactively viewing and zooming into the execution at an extremely fine granularity (e.g., sub-nanosecond level).

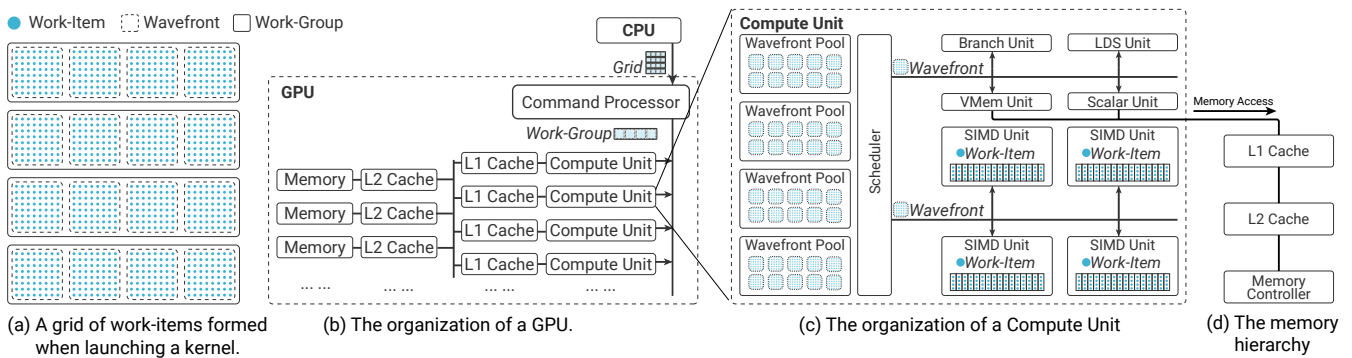


Figure 2: An overview of the GPU execution model and hardware organization.

Moreover, Compute Units read data from memory (see Figure 2 (d)). After data is copied from the CPU, it is stored in the GPU’s main memory. Due to physics limitations, the main memory is too slow to feed the data required by the Compute Units. To increase throughput, GPU designers place faster, but smaller, storage close to Compute Units. Caches are organized in tiers, labeled as “Level 1” (L1) or “Level 2” (L2) caches, with the L1 caches being smallest, fastest, and placed closest to the Compute Unit. In many applications, the memory system implementation limits the performance of the whole GPU. Visually examining the memory system is crucial for understanding overall GPU performance.

GPU simulators. GPU simulators [BYF*09, UJM*12, GUK17, SBM*19, KSAR20] recreate hardware behaviors with just a CPU. Hardware designers can run simulators and gather key performance metrics (e.g., execution time, number of instructions executed). A simulator can also dump details (i.e., traces) to a file to record all the events that occur during a simulation. Since the trace is highly detailed, simulating a millisecond GPU execution can generate several GBs of trace data.

3. Related Work

Performance visualization tools can help engineers to identify bottlenecks and make evidence-based performance-improvement decisions. Here we survey several such tools.

Call-graph visualizations. Many visualization tools for debugging performance issues in CPU programs rely on call graph data. These tools can be categorized into three groups: 1. Icicle plots [KL83] or flame diagrams [Gre16] are commonly used to show the hierarchical relationship between function calls and their execution time. Example tools include perf [Per], gperftools [gpe], callgrind [Val], Chrome Developer Tools [chr20], and Very Sleepy [Ver]. 2. Node-Link visualizations (e.g., pprof [ppr], kcachegrind [KCa]) show function call relationships and use the size of nodes and links to visualize execution time. 3. Pipeline-based visualization tools use multi-stage Gantt-chart [Cla22] to show function executions (e.g., TraceVis [RZ04]). These tools were mainly developed for single-threaded CPU execution and cannot easily be used to visualize complex multi-threaded behavior.

Recently, new tools are developed to visualize multi-threaded executions. Isaacs et al. [IBJ*14, IBL*15] used logical times to align and compare the communication and dependency patterns between

threads. Perfopticon [MHHH15], specifically designed for evaluating SQL query performance, could identify issues in query execution. Perfopticon used nested Gantt charts to represent the relationship between nested SQL queries.

Existing software-oriented performance visualization tools lack the proper abstractions needed to visualize GPU hardware execution. Hardware executions involve high degrees of parallelism, though without function calls, demanding new data abstractions and new visualization strategies. To address this research gap, our work aims to bridge event-based data representations (typically used to represent hardware execution) and call-graph-based data representations by building an abstraction of hardware behavior leveraging hierarchical tasks.

Visualizing parallel distributed computing. Most visualizations frameworks targeted parallel computing performance focus on large-scale distributed systems, and highlight the communication between computing nodes. Bhatele et al. [BGI*12, ILG*12] developed Boxfish to visualize the multi-dimensional node organization on parallel systems, capturing the communication patterns between the nodes in a parallel distributed computing environment. Ross et al. [RWP*19] presented a tool that visualized not only the network traffic but also the behavior of the parallel simulation that models the communication. ChampVis [PGN*19] provided a solution to compare different hardware executions. The capabilities of distributed-system network visualization have been extended recently with automatic route suggestions [FMR*17] and anomaly detection [XXM18, BLX*20]. However, hardware visualization poses different challenges (e.g., sub-nanosecond visualization, large number of component types), and hence, existing solutions cannot be directly used.

GPU performance visualization. GPU vendors have created profilers to enable developers to examine GPU performance (e.g., the Radeon GPU Profiler [rad20] and the NVIDIA Visual Profiler [Cor14]). Academia has also developed visualization tools to understand CPU-GPU heterogeneous execution [GPMSS*18]. However, these tools mainly focus on high-level performance metrics in GPU software development and cannot show sub-nanosecond hardware execution details. Due to the lack of detailed GPU execution information, it is challenging for hardware designers to use industrial profilers to examine hardware behaviors.

There are a few tools developed specifically for visualizing GPU hardware execution. Rosen [Ros13] proposed a solution to visu-

ally inspect how GPUs access shared and global memory during CUDA kernel executions. AerialVision [AFTA10] provides a time-lapse view for tracking the changes of several performance metrics. It presents a view of the statistics associated with individual lines of source code to help identify the root cause of bottlenecks. M2S-Visual [ZUSK15] was customized to visualize execution traces generated by Multi2Sim [UJM*12]. However, these existing tools lack the ability to easily compare metrics across hardware components. New visualization solutions are desired to simplify GPU performance analysis.

4. Task Analysis and Abstraction

To characterize the workflow of GPU system performance analysis, we conducted a survey study and interviewed 5 GPU hardware designers (none of them are authors of this paper, upon approval from our Institutional Review Board). Below, we describe the processes of the survey study, interview sessions, and the domain characterization including data, task analysis and abstraction.

Study design and participants. We conducted a survey study with 37 computer hardware designers (none of them are authors of this paper) with experience analyzing parallel computing system performance. We created the survey using Google Forms (see supplemental materials) which was distributed via social media (e.g., Twitter, Slack). The survey contained three sections. **The first section** aimed to understand users' prior experience with performance analysis, such as how often respondents analyze computing system performance and use simulators during their daily work, what level of difficulty they experience when identifying and making sense of performance issues, and how much they feel that they need tools to help them examine performance issues. This section contained seven items using a five-point Likert-style scale, with choices ranging from (1) Strongly Disagree to (5) Strongly Agree. **The second section** aimed to help us characterize workflows and strategies. Four open-ended questions were asked to understand what simulators they have used and the tools that help them identify performance issues, and to examine their process for identifying performance issues. **The last section** included five questions regarding respondents' positions and affiliations, length of working on performance analysis, and whether or not they would be willing to be contacted for clarification questions and follow-up studies. Participants had various working experience on performance analysis: <1 year (n=7), 1–2 years (n=6), 2–3 years (n=6), 3–5 years (n=9), 5–10 years (n=7), and >10 years (n=2).

We also conducted semi-structured interviews with 5 GPU hardware designers to understand their daily practices of performing GPU performance analysis. The goal was to further refine the design requirements and characterize practices and common workflows when using simulators. The first two authors open-coded [Tho06] the survey responses and interview transcripts, and identified themes iteratively through discussion and affinity diagramming that helped us organize the codes.

The aim of the survey and interview was to examine the workflow of carrying out performance analysis and guide us to help characterize domain tasks and processes. However, we are aware that the survey results may have potential biases since we used conven-

ience sampling for recruitment (e.g., under-representation of sub-populations in the sample). Therefore, we avoid making absolute claims when analyzing and reporting our survey results, necessitating care when interpreting the results.

Task analysis. Based on the survey results and interviews, we performed a task analysis to characterize domain problems followed by task abstraction that aims to recast user goals from domain-specific languages to a generalized terminology for better understanding and generalizability [Mun14] (see Figure 3). We chose a Hierarchical Task Abstraction [ZCD19] as performance analysis in hardware design is a dynamic process that typically involves multiple hierarchical abstractions.

The overall goal of GPU hardware designers is to improve GPU performance by proposing new GPU hardware designs (Task 0). The first step is to evaluate the current design's performance (Task 1). In the first iteration, designers evaluate the performance of the baseline design by running a simulation (Task 1.1) and collecting performance metrics and GPU execution traces (Task 1.2).

After collecting the execution traces, GPU hardware designers need to identify performance bottlenecks (Task 2), checking if the components are overloaded or underloaded (Task 2.1). To complete this, designers need to find the component (Task 2.1.1), select the metrics to examine (Task 2.1.2), and compare with anticipated values (Task 2.1.3). After identifying the load on a component, the designer needs to find the root cause for each underloaded or overloaded component (Task 2.2) by comparing directly connected components (Tasks 2.2.1) and parallel components (Task 2.2.2).

To explore a hypothetical performance bottleneck, the GPU hardware designer typically examines execution details during a short time period (usually under 0.1 microsecond, Task 2.3). The designer will first check the tasks that are executing in a particular hardware component (Task 2.3.1). For example, they check all the instructions executing in a Compute Unit to see which instructions are preventing the Compute Unit from making further progress. They also check the task hierarchy (Task 2.3.2). For example, a designer may want to check which memory accesses are conflicting with a memory load instruction.

After identifying the bottleneck of the current platform, the GPU hardware designers need to propose new designs that can mitigate the performance bottlenecks (Task 3). This step mainly relies on problem-solving skills and prior experience, thus it is out of the scope of this work. After developing a new design, designers need to repeat Task 1. This time, however, they need to compare the result with the baseline design (Task 1.3) and examine whether the bottleneck is resolved as expected (Task 1.4). If the performance of the new design cannot reach the performance goal, designers will look for a new performance bottleneck in the new design (Task 2). Designers may need to repeat Tasks 1–3 many times until they find a solution that satisfies the design requirements. Due to the page limitation, more detailed task analysis with examples are relegated to supplemental materials.

Task abstraction. Based on the task analysis, we performed a task abstraction that aims to recast tasks from domain-specific languages to a generalized visualization terminology to achieve better understanding and readability [Mun14]. We generate task abstrac-

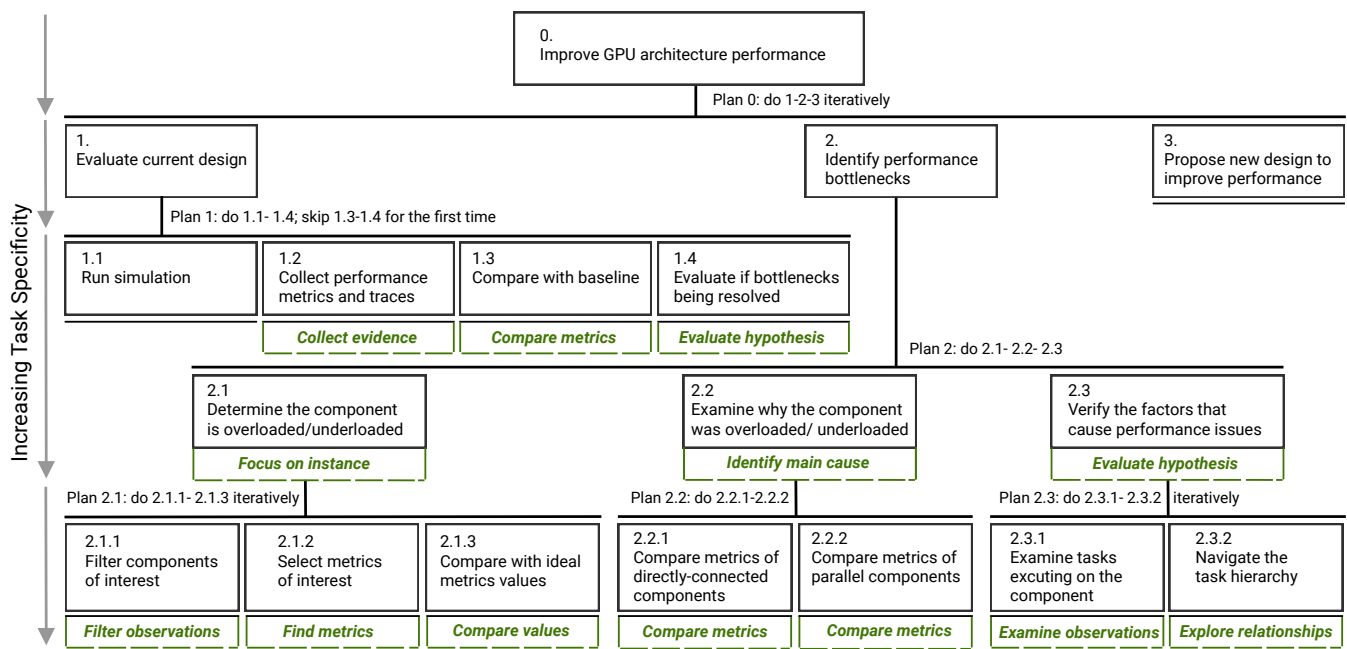


Figure 3: Hierarchical Task Abstraction [ZCD19] of GPU hardware performance analysis: Each task is shown as a box with an ID that indicates the associated level of the hierarchy. The vertical lines connect the tasks and their subtasks. Labels on the vertical lines (i.e., plan) are used to identify the task sequences. Horizontal lines that sit beneath the corresponding boxes mean that there are no subsequent subtasks. Task abstractions derived from the tasks are highlighted in dashed green boxes Task abstraction.

tions guided by Amar et al.’s [AES05] low-level components of visual analytics and Lam et al.’s framework [LTM18] that focuses on connecting analysis goals and steps in design studies. The abstractions, labeled green in Figure 3, are integrated with the Hierarchical Task Abstraction guided by Zhang et al.’s work [ZCD19].

Design requirements. We identified a set of design requirements (DR) of GPU execution trace visualization based on the task analysis. **DR1:** Support visualizing a large number of concurrently-executing tasks. Hardware elements typically achieve a high degree of parallelism. Thousands of components may execute different tasks at the same time. Each component may also execute a large number of tasks in parallel. **DR2:** Create a mapping between software concepts and hardware concepts. It is essential to examine how the software elements (e.g., memory access) are executed on hardware components (e.g., caches). If designers question why a software element is not executed early enough, they need to investigate why the hardware elements are busy. **DR3:** Allow for efficient browsing through the execution and zooming down to focus on a short duration gradually. Simulation traces record executions that last from milliseconds to seconds. Meanwhile, researchers need to identify hardware behavior at a sub-nanosecond level.

5. Data Abstraction

We first map data into a unified format. The goal is to build a cohesive and comprehensive data collection framework to help users make sense of the volumes of data and identify performance issues. We elect not to use prior parallel computing log formats [KBB*06, KBD*08, CGL00] because: (1) Existing formats trace events rather than tasks. Pairing start-task events with end-

Table 1: A summary of data abstraction

Field	Format	Description	Example
ID	Text	The unique identifier of a task, which was randomly generated to guarantee uniqueness	5C9dX8
Parent ID	Text	The ID of the parent task	7F3sY2
Category	Text	The category of the task belongs to	Instruction
Action	Text	The job of the task	Read Memory
Location	Text	The hardware component that carries out the task	CPU1.Core1
Start/ End	Time	The time that a hardware component starts to process/ completes processing the task	0.00014566s

task events requires extra processing. (2) It is difficult to capture the hierarchical relationship of the tasks with existing formats. (3) While the design of prior formats considered query performance, it is challenging to use when working with file-based traces. Using a database program can address this issue and provide low-latency query services for a wide range to query types.

We summarize all the trace entities as tasks. Each task contains multiple properties, including ID, parent ID, category, action, location, start/end time, and details (see Table 1). Each task is assigned a unique ID. A task is also assigned information describing which category it belongs to and the job of the task, captured in the “Category” (e.g., “Request Out”, “Request In”, “Instruction”) and “Action” (e.g., “Read Memory”, “Execute ADD Instruction”) fields, respectively. The name of the component that carries out the task is recorded in the “Location” field. Each task has a Start time and an End time. Tasks are organized hierarchically using a tree structure. Each task has a “Parent ID” field that tracks a task’s parent task, except for the root task representing the entire simulation.

We record component-to-component communication with two special task categories: (i) “Request Out”, and (ii) “Request In”.

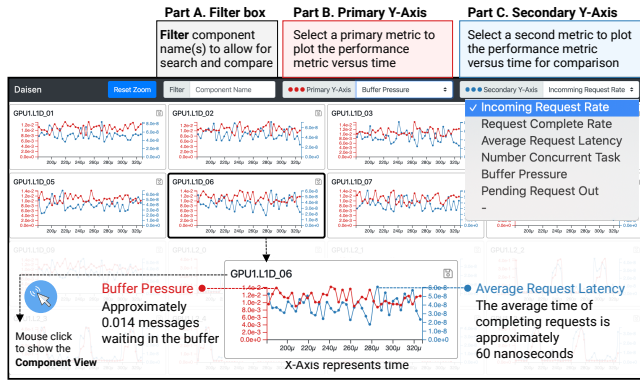


Figure 4: The Overview Panel shows multi-dimensional data. Users can filter components (Part A) using regular expressions and customize the display of dual y-axes (Parts B&C).

Suppose a Compute Unit wants to read data from an L1 cache, the Compute Unit would initiate a “Request Out” task when the memory read request is created. When the L1 cache receives the request, a “Request In” task is created, setting the “Request Out” task as the parent task. After the L1 cache reads the data, either from its local storage or from an L2 cache, the L1 cache sends the data back to the Compute Unit and ends the “Request In” task. Finally, when the Compute Unit receives the response, the Compute Unit ends the “Request out” tasks. By tracking the “Request In” and the “Request Out” tasks, we can fully capture the communication between the components.

6. Visualization Tool

Next, we introduce Daisen’s visualization tool, an open-source, web-based, interactive visualization tool that aims to help GPU hardware designers evaluate the performance of GPU designs. We highlight its capability of fulfilling the requirements with three unique perspectives, including the Overview Panel, the Component View, and the Task View.

6.1. Overview Panel

The Overview Panel (see Figure 4) enables users to see the big picture of the execution and locate interesting time intervals that require closer investigation — partially fulfilling DR-3. The Overview Panel displays a series of small multiples, with one diagram per component. We separate the large number of GPU components into multiple pages.

We provide a *filter box* (see Figure 4 Part A) to enable users to filter the components of interest, satisfying the requirement of targeted expert users. The filter box also supports task 2.2 (compare metrics of related components). Users can use regular expressions (e.g., "(CU|L1|L2)") to create the combination of the components to compare.

We display all the component diagrams as time-series visualizations, with the x-axis indicating the simulation time. Users can select key metrics of interest to explore using a drop-down menu. They can also choose the metrics for both the primary y-axis

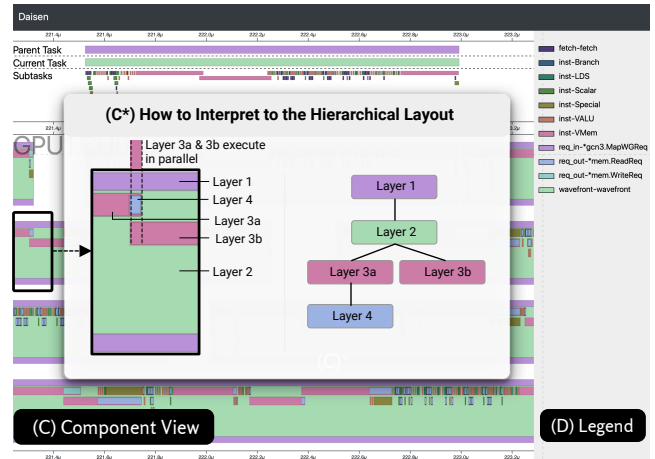


Figure 5: The Component View (C) hierarchically shows all the tasks executed in a given component. (C*) illustrates how the layout in the Component View maps to the tree structure. The legend (D) shows the color encoding for the tasks, following the format of Category-Action. (Note: C* is not displayed in the interface.)

(see Figure 4 Part B) and the secondary y-axis (see Figure 4 Part C). We currently provide 6 metrics for the y-axes, selected according to the survey responses and feedback from domain experts. Each data point represents the average value for the metric over a small time window. Although averages may hide outliers, they are effective for identifying general trends, which is in line with the goal of the Overview Panel.

When a user zooms or drags in a diagram, the time axes of all the diagrams will be updated to show the curves of the same time period. This creates an alignment effect and can help users compare the behavior of different components [ZDBS* 19]. Once the user arrives at an initial guess for the cause of the performance bottleneck, they may need more detailed information to verify the performance bottleneck, as in Task 2.3. The user can first zoom into a short time period in the Overview Panel and click the component name to see the tasks executing during that period in the Component View.

6.2. Component View

The Component View, as shown in Figure 1(C), shows all the tasks executed at a certain component. The Component View aims to help users understand how hardware resources are utilized. The current component name (using transparent text) on the top-left corner of the Component View helps remind users which component they are currently viewing.

Horizontal placement. We use Gantt-like charts to represent tasks in the Component View, which is consistent with existing trace visualization tools [RZ04, chr20], flattening the learning curve and reducing mental burden [Rog11].

We choose to present the task visually in a hierarchical fashion to avoid clutter and allowing users to clearly understand the relationship between the tasks (see Figure 5). In the Component View, root tasks (a task that has no parent task or its parent tasks have a

different location) are directly displayed on the canvas. Each task uses its internal space to show its subtasks.

Vertical placement. We develop an up-floating layout algorithm to determine where to place the task bars on the y-axis. We assign the row number for each task in the order of the task start-time. Each task is assigned with the smallest row number (top-most row), as long as it does not overlap with any task that is already assigned with that row number. In this way, we can use the minimum number of rows and show the maximum height possible for each task, while guaranteeing that tasks do not overlap with each other.

We maximize the bar height to best utilize the vertical space. However, the bar heights are constrained by two conditions. First, no two bars should overlap to ensure readability. Second, we regulate the bar height so that same-level bars have the same height, to maintain logical consistency.

Users can rely on the number of rectangles stacked at the same x-position, indicating how many tasks are executing in parallel, on a specific component, at a particular time. This shows how busy the component is. The blank spaces on the y-axis are results of the bar height regularization, and should not be misinterpreted as the component being underutilized. We allow y-axis scrolling if there are too many tasks that are executing concurrently.

Color coding. We use color-coding to differentiate tasks based on their “Category” and “Action”. Since we cannot exhaustively list all the possible “Category” and “Action” values a priori, we need to dynamically color the tasks. To solve this problem, we use the CubeHelix [Gre11] coloring scheme, as CubeHelix creates contrast in both the hue and the lightness, ensuring readability and accessibility for colorblind users. When there are too many “Category”–“Action” pairs, we fall back to code “Categories” to avoid using too many colors.

One drawback of using a dynamic color-coding solution is that the colors can change as the user interacts with the tool. E.g., when a user zooms out, more tasks will appear on the screen, using more “Category”–“Action” pairs. To differentiate the tasks, we need to assign more colors to the tasks. The current recoloring solution changes the colors of tasks that are already on the screen and may add extra mental burdens for users. A possible solution is to lock the color of the tasks already on the screen and generate new colors for newly appeared “Categories” and “Actions”. We will leave improving the dynamic color-coding solution as future work.

Interactive legend. We reserve the right sidebar of the interface to provide information that may help users understand the diagrams. By default, color-coded information is displayed as a legend. The legend also allows users to find tasks of a particular type easily. When a user hovers the mouse pointer on a legend element, the tasks in the Component View, as well as the tasks in the Task View, will be highlighted with bolder outlines, while the rest is grayed out. The right sidebar can also provide additional information for each individual task. When the user hovers their mouse pointer on a particular task in the Component View, the detailed information (listed in Table 1) will be shown above the legend.

6.3. Task View

The Component View demonstrates how the hardware is utilized. It aligns with a typical hardware performance analysis scenario. However, users also need to analyze the performance using a software performance analysis approach. They usually need to answer the question about why a particular task takes such a long time or which subtask takes the longest portion of the total time. Although the Component View hierarchically displays tasks, it cannot show subtasks that are not executed in the current task, and hence, does not allow the user to move up or down the entire task hierarchy, as required in Task 2.3.2. To solve these problems, we supplement the Component View with a Task View. A user can click on a task in the Component View to enable the Task View. The selected task becomes the “current task”.

As shown in Figure 1 (B), the Task View shows three groups of tasks. The second row of the Task View shows the “current task”. Above the current task is its parent task. We show the parent task for two purposes. First, users can understand the relative timing relationship between the current task and the parent task, answering the question of whether the current task is a major step required to complete the parent task. Second, users can click on the parent task to navigate towards the root of the task hierarchy, getting a larger picture of program execution and partially fulfilling DR-3. All the subtasks of the current task are displayed below the current task. We reuse the “up-floating” algorithm to maintain consistency with the Component View and to save space. Showing the subtasks allows users to discover how the time is spent to complete the current task. Users can click on one of the subtasks to set that task as the “current task” or to drill down for a more detailed view. Combined with the parent task, the subtasks support navigation up and down through the entire task tree, fulfilling DR-3.

The Task View is stacked on top of the Component View. Their time axes are always aligned. Dragging or zooming one view always refreshes both views. Aligning the time axes helps the user to easily match the tasks in the Task View (software concept) and the Component View (hardware concept). When a user hovers over a task in either the Task View or the Component View, the task is highlighted on both sides to help users establish the connection between these two views. A typical use case is that a user may need to know why a task is not executed immediately after the previous task is completed. To answer this question, the user can check the task that occupies the component earlier. These two tasks are competing for hardware resources. As we align the Task View and the Component View, it should be straightforward to find the tasks that the component is executing at the time. The alignment of the Task View and the Component View can fulfill DR-2.

6.4. Implementation

We provide a data collection library — an extremely light-weight one with only 7 functions defined — that can be invoked from the simulator to export trace data. Minimum changes are required to record the tasks. For example, we instrumented the MGPUSim’s L1 cache model by adding 16 lines of code, as compared to the ≈ 3500 lines of code of the original L1 cache model. We have fully instrumented MGPUSim [SBM*19] and our instrumentation


is a standard part of MGPUSim releases. We select to instrument MGPUSim because it is flexible and is widely used in state-of-the-art GPU architecture research projects [BSM*20, MSD*20a, MSD*20b, BSD*20, LSJ*19, TSAK19].

The visualization tool is implemented with the client-server model. Users can start the backend server during the simulation execution or after the simulation completes. The server application reads data from the database, processes the data, and sends the data to the frontend upon request. The frontend was implemented using TypeScript and D3.js. This server-client model does not require the installation of any library or software except for a web browser on the client side. The server-client model also enables multiple users to check the results at the same time.

7. Case Study

We use a case study performed by a domain expert (an author of this paper) to demonstrate how Daisen can help improve the performance of a GPU architecture design. We simulated the execution of the PageRank benchmark from the Hetero-Mark benchmark suite [SGZ*16] running on an AMD R9 Nano GPU [AMD15]. We first ran the PageRank benchmark [Ber05] on an R9 Nano GPU with MGPUSim. MGPUSim reported the program could run in $\approx 51ms$. The simulation generated a trace with $\approx 32M$ tasks, consuming $\approx 6.1GB$ disk space.

The first task was to check which component was underloaded or overloaded to examine if the workload was compute-intensive or memory-intensive. To check compute intensity, we first checked the utilization of the SIMD unit (digital circuits that run the calculations) by filtering the SIMD unit by entering “SIMD” in the filter box and choosing “Number of Tasks” as the primary y-axis in the Overview Panel. According to Figure 6(a), the SIMD utilization was low, suggesting it was not a compute-intensive workload. We then checked the “Buffer Pressure” and the “Request Completion Rate” for the L1 and L2 caches. From Figure 6(b) and (c), we were able to conclude that both the buffers of the L1 and L2 caches were empty for most of the time and the Buffer Pressure value was constantly smaller than 1. We can conclude that this work was not memory intensive either. Since the L1 cache was not given enough tasks (as suggested by the low buffer pressure), we further examined the Component View by clicking on the L1 cache widget title. In the Component View, we identified long gaps between tasks, indicating the L1 cache is idle for most of the time. After this, we attempted to make sense of why the L1-Cache tasks did not start earlier. To reason about the process, we selected the second task to enable the Task View. We kept using the “parent task” bar in the Task View to trace back to the root of the tree.

As we traced back to the Compute Unit, we could check the memory instructions that triggered the L1 cache access. We could identify that the reason why the L1 did not have enough tasks was that the memory instructions were executed infrequently. Ideally, each Compute Unit can run 40 wavefronts  (as shown in Figure 6(d)), so that there should be a large probability that at least one wavefront issues a memory instruction. However, in Figure 6(d), we can see that only 3-5 wavefronts were executing in parallel. These wavefronts might be all executing compute instructions or

waiting for existing memory accesses to complete, so they cannot generate new memory accesses. After making sense of the performance issue, we concluded that the root cause might lie in the Compute Unit that was not executing enough wavefronts. The global performance bottleneck was that the Command Processor was not running fast enough to dispatch work-groups.

A solution to the problem above is to increase the dispatching speed. As we change the dispatching speed from one work-group per cycle to two work-groups per cycle, MGPUSim reported that the GPU execution time is reduced from $\approx 51ms$ to $\approx 44ms$, speeding up the execution by $1.16\times$. If we compare the work-group and wavefront execution in Figure 6 (e) and (f), we can see that the Compute Unit had been executing more work-groups at the same time (see the 6th row). Overall, the findings from Daisen, and the minor modification of the GPU design, improved device utilization, and reduced the overall execution time.

8. Evaluation

To further evaluate Daisen, we conducted user studies with 10 GPU hardware designers (9 of the 10 also submitted the survey mentioned above; none of the 10 are authors of this paper). The average time they had worked in performance analysis was 4.6 years ($sd = 2.8$). The highest degree of education among participants was split equally between master’s ($n = 4$) and doctoral degrees ($n = 4$), with one bachelor’s and one associate’s degree.

8.1. Methodology

We conducted our evaluation study remotely using online communication tools (e.g., Zoom). We asked participants to share their screens to allow us to observe how they interacted with the visualization framework. The evaluation study includes five parts. (1) Participants were provided with a walk-through tutorial. (2) They were asked to explore the tool on their own using the “think-aloud” method [Lew82] that encourages verbalizing their thoughts as they move through the visualization. In this process, participants were asked to explore a Matrix Transpose benchmark (1024×1024 matrix, a well-known benchmark for GPU computing) running on an R9 Nano GPU. The dataset has a similar scale as compared to the dataset used in the case study. (3) They were asked to identify any performance bottlenecks they examined. (4) We conducted follow-up semi-structured interviews about their experience interacting with Daisen, accessing ease-of-use, what users liked and disliked about Daisen, and what changes they would make to help them identify performance issues more efficiently. (5) We asked each participant to fill out a post-study survey (see Figure 7 and supplemental materials for details) regarding how well Daisen facilitates performance analysis. The whole session lasted about 90 minutes for each participant.

8.2. Findings

Our participants gave overall positive feedback on Daisen, especially how Daisen can be used to help them identify and make sense of performance issues. During the evaluation, 9 out of 10 participants successfully identified that the benchmark was a memory-

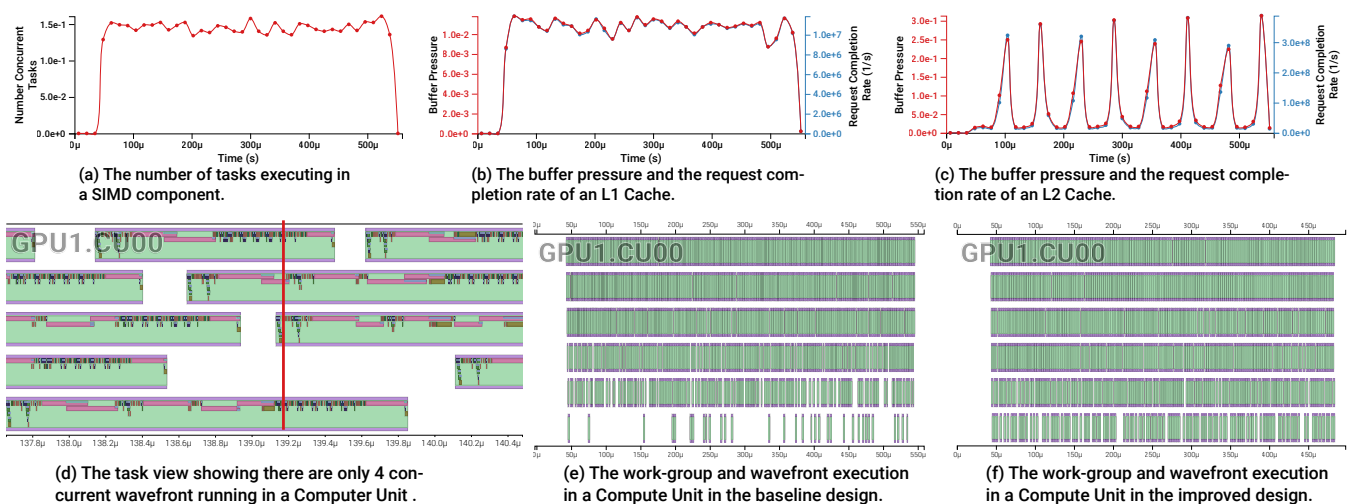


Figure 6: Example screenshots that help with workload characterization in the case study.

intensive benchmark. 7 out of 10 participants pointed out that reducing the latency or increasing the bandwidth of the caches and the memory controllers can improve performance.

Value of visualizing GPU traces in Daisen. Participants appreciated the GPU simulation visualizations and some expressed excitement about using Daisen to examine GPU performance issues. They also described how they typically visualize execution information. Half indicated that they first added “print functions” in the GPU simulator to get more performance information, exported to a CSV file and then visualizing the trace data separately (in Python) to explore and identify performance issues. E.g., P03 said “*My debugging is using prints to get any of these numbers. But is also cumbersome to either have another script to process those files and then read them.*” P03 further appreciated Daisen’s capability of providing visual clues related to performance issues. P03 said “*Daisen gives a pretty good picture of this benchmark is suffering because of certain factors. Then I can take a design step to improve something.*” P07 also liked how Daisen supports visualizing GPU traces, comparing it with VTune [Rei05] and nvprof [nvp20], other trace visualization tools. P07 mentioned that “*nvprof also gives you some kind of views but can only show which kernels are running (from a high-level and coarse-grained perspective). However, Daisen enables users to understand which portion and phases of the application are getting low [performance] and have problems.*”

Overview allows quick hotspot identification. 7 out of 10 participants started to use the filter box to narrow down the components after browsing no more than 4 pages. None of the participants attempted to check all pages. Moreover, 9 out of 10 participants displayed little difficulty finding the desired components by typing in commonly-used computer architecture terms such as “L1” or “TLB”. The participants’ way of interacting with the Overview Panel suggests our participants (as domain experts) can easily learn how to use Daisen to solve problems.

Participants also frequently zoomed in and out to search hot spots that took longer in execution. P05 specifically valued being able to browse behaviors of various components, as “*it gives the*

users the ability to verify their hypothesis on different applications and see which resources cause the constraint in the system.”

Task View enables navigation between components. Our participants used the Task View mainly to navigate across components and explore execution relationships between the Task View and the Component View. P04 and P08 particularly liked the Task View’s ability to navigate between components. P08 said “*It is really really nice to have this type of navigation between tasks and subtasks.*”

Component View helps examine low-level behaviors. Our participants mainly used the Component View to get a more in-depth understanding of the performance of a particular component, especially when they saw conflicting things happening in an overview. For example, P03 recalled a situation where he needed a detailed view. He explained that when he tried to understand what instructions the workload was executing, he needed more evidence to support his hypothesis on how to improve the design with the help of the Component View. Similarly, P01 specifically mentioned how well Daisen helped him understand simulation behavior as compared with some profilers he had used before. He commented that existing profiling tools failed to provide sufficient details for making sense of performance issues.

Though all participants foresee themselves using Daisen to facilitate their research in GPU hardware design and computer architecture research, they suggested a few things that can be improved to further help them examine GPU performance issues. These features include (1) educational support and tutorings, such as automatic screen recording and more user guidance, and (2) integration of summary statistics into Daisen to enable users with various levels of visualization literacy to understand the performance (e.g., statistically summarizing 70% of the instructions being executed).

Post-study survey results. Overall, as shown in Figure 7, participants found that using Daisen helped them understand key characteristics of workloads (median 7, IQR 0.75), identify performance issues (median 7, IQR 1.5), locate overloaded hardware components (median 7, IQR 1), explore trends in executions (median 6, IQR 1), examine software-related factors that cause the performance bottlenecks (median 5, IQR 2), identify opportunities to im-

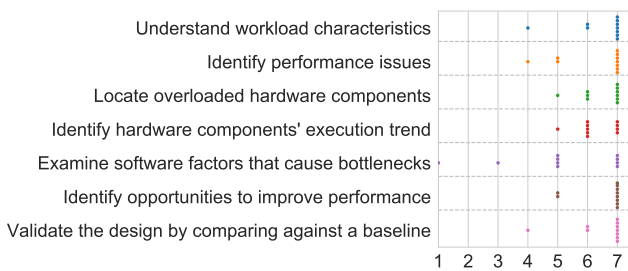


Figure 7: Post-study survey results: Participants rated whether they were able to perform a set of tasks using a 7-point scale, labeled from Strongly Disagree (1) to Strongly Agree (7).

prove performance (median 7, IQR 0), and validate the new design by comparing with a baseline (median 7, IQR 0.75).

9. Discussion

Participant feedback about Daisen was very positive overall. In this section, we reflect on our task analysis, design, and evaluation.

Our task analysis and abstraction was drawn from our survey and interview studies with GPU designers. As we described, some tasks are particularly essential for GPU performance analysis, such as comparing metrics of parallel components (Task 2.3.2). Some other tasks may be applicable to other sibling domains (e.g., designing CPUs and domain-specific accelerators); thus, we suggest future research to adapt the Daisen design to fit design requirements for other types of digital circuit devices.

Generalization vs. specification. One challenge we encountered when designing Daisen was to balance between generalization and specialization. On the one hand, we can pursue generalizability that aims to benefit a wider range of audiences [TLCC17]. On the other, design studies have focused on seeking solutions for a particular problem domain [Mun08]. Daisen's design avoids a close binding between the visualization style and the component to be examined. Instead, we use the same views to reveal the behavior of most of the components on a GPU, such as Compute Units, caches, and memory controllers.

Our effort to maintain a general framework created some challenges. Our participants felt that the component sorting in the Overview Panel was not intuitive, and that it took some time for them to understand how components were connected. Although most participants indicated that they were able to infer the component-specific metrics from the component-independent metrics, some (P03, P07) also wanted to see traditional component-specific metrics to help them understand the execution. Therefore, more work is needed to examine the best trade-off between generalization and specialization. For example, we plan to allow users to tag the tasks with user-specified properties and use the tags to visualize component-specific metrics.

GPU visualization for non-expert users. We noticed that more experienced GPU designers more quickly grasped key concepts visualized in Daisen. This shortcoming raises a question for future research: how to design a visualization environment for users with different levels of expertise and help overcome the steep learning curve. For example, non-experts may not be able to pinpoint

interesting components in the Overview Panel and may end up browsing pages. Many approaches have been explored to help users comprehend how visualization works, such as step-by-step wizards, forums, and animated video tutorials [FGF11, GF10, MGF11, PDL*11]. In addition to these strategies, we plan to add better navigation support (e.g., grouping, ordering) in the Overview Panel and provide visual hints about how the components are connected. We also plan to have each component specify the expected metric values, making it easier for beginners to compare metrics.

Limitations. There are several limitations in our current design which we are planning to improve. While abstracting hardware execution behavior can represent most digital circuit behaviors, some corner cases may not perfectly apply. E.g., DRAMs usually spontaneously and periodically initiate a refresh action, which does not belong to the task hierarchy tree. We currently do not capture such behavior but will generalize to a graph data representation. Another limitation is that we do not highlight communication between GPU hardware components. In chip design, on-chip network bandwidth is a scarce resource and can limit overall performance. We plan to add a network view and time-lapse traffic visualization. Moreover, we currently do not support comparing executions between different benchmark executions and different hardware configurations. Finally, the current design requires a good knowledge of computer architecture and information about the simulation being visualized. Future work also includes using heuristic algorithms as well as narrative solutions to highlight potential bottlenecks.

10. Conclusion

Visualizing massively parallel GPU hardware execution has been challenging due to the large number of digital circuit components and the complex interplay between them. To address this problem, we have described Daisen, which includes a dedicated data representation for simulator-generated hardware executions and a web-based interactive visualization tool. Using a case study and user evaluation studies, we demonstrated how Daisen can be used to help GPU hardware designers identify performance bottlenecks. Daisen is currently under active development, with more features and improvements being added to the tool. We hope the design of Daisen can serve as the first step of a series of works that can help hardware designers build explainable computer architecture, since only human-understandable computer architecture can avoid costly hardware bugs, performance bottlenecks, reliability issues, and security vulnerabilities.

Acknowledgments

We thank our reviewers for their constructive feedback. We also thank AMD for supporting this work.

References

- [AES05] AMAR R., EAGAN J., STASKO J.: Low-level components of analytic activity in information visualization. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* (2005), pp. 111–117. doi:10.1109/INFOVIS.2005.24. 5
- [AFTA10] ARIEL A., FUNG W. W., TURNER A. E., AAMODT T. M.: Visualizing complex dynamics in many-core accelerator architectures.

- In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)* (2010), pp. 164–174. doi:10.1109/ISPASS.2010.5452029. 2, 4
- [AMD15] AMD: AMD Radeon R9 Nano, world's smallest and most power-efficient enthusiast graphics card, brings 4K gaming to the living room, 2015. URL: <https://www.amd.com/en/press-releases/amd-radeon-r9-nano-2015aug27>. 2, 8
- [ANM*12] ARORA M., NATH S., MAZUMDAR S., BADEN S. B., TULLSEN D. M.: Redefining the role of the CPU in the era of CPU-GPU integration. *IEEE Micro* 32, 6 (2012), 4–16. doi:10.1109/MM.2012.57. 2
- [Ber05] BERKIN P.: A survey on PageRank computing. *Internet Mathematics* 2, 1 (2005), 73–120. doi:10.1080/15427951.2005.10129098. 8
- [BGI*12] BHATELE A., GAMBLIN T., ISAACS K. E., GUNNEY B. T., SCHULZ M., BREMER P.-T., HAMANN B.: Novel views of performance data to analyze large-scale adaptive applications. In *SC'12: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), pp. 1–11. doi:10.1109/SC.2012.80. 3
- [BLX*20] BESCHASTNIKH I., LIU P., XING A., WANG P., BRUN Y., ERNST M. D.: Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–38. doi:10.1145/3375633. 3
- [BSD*20] BARUAH T., SUN Y., DINÇER A. T., MOJUMDER S. A., ABELLÁN J. L., UKIDAVE Y., JOSHI A., RUBIN N., KIM J., KAELI D.: Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), IEEE, pp. 596–609. doi:10.1109/HPCA47549.2020.00055. 8
- [BSM*20] BARUAH T., SUN Y., MOJUMDER S. A., ABELLÁN J. L., UKIDAVE Y., JOSHI A., RUBIN N., KIM J., KAELI D.: Valkyrie: Leveraging inter-tlb locality to enhance gpu performance. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (2020), pp. 455–466. doi:10.1145/3410463.3414639. 8
- [BYF*09] BAKHODA A., YUAN G. L., FUNG W. W., WONG H., AAMODT T. M.: Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software* (2009), pp. 163–174. doi:10.1109/ISPASS.2009.4919648. 2, 3
- [CGL00] CHAN A., GROPP W., LUSK E.: Scalable log files for parallel program trace data draft. *Argonne National Laboratory, Argonne, IL 60439* (2000), 1–61. 5
- [chr20] Google Chrome - the new Chrome & most secure web browser, 2020. (accessed on 03/19/2020). URL: <https://www.google.com/chrome/>. 3, 6
- [Cla22] CLARK W.: *The Gantt Chart: A Working Tool of Management*. Ronald Press, 1922. 3
- [Cor14] CORPORATION N.: NVIDIA Visual Profiler–NVIDIA Developer Zone, 2014. 3
- [FGF11] FERNQUIST J., GROSSMAN T., FITZMAURICE G.: Sketch-Sketch revolution: An engaging tutorial system for guided sketching and application learning. In *Proc. 24th Annual ACM Symposium on User Interface Software and Technology* (2011), UIST '11, p. 373–382. doi:10.1145/2047196.2047245. 10
- [FMR*17] FUJIWARA T., MALAKAR P., REDA K., VISHWANATH V., PAKA M. E., MA K.-L.: A visual analytics system for optimizing communications in massively parallel applications. In *2017 IEEE Conference on Visual Analytics Science and Technology* (2017), VAST, pp. 59–70. doi:10.1109/VAST.2017.8585646. 2, 3
- [GF10] GROSSMAN T., FITZMAURICE G.: ToolClips: An investigation of contextual video assistance for functionality understanding. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* (2010), CHI '10, p. 1515–1524. doi:10.1145/1753326.1753552. 10
- [GKM82] GRAHAM S. L., KESSLER P. B., MCKUSICK M. K.: Gprof: A call graph execution profiler. *ACM Sigplan Notices* 17, 6 (1982), 120–126. doi:10.1145/872726.806987. 2
- [gpe] gperftools/gperftools: Main gperftools repository. <https://github.com/gperftools/gperftools>. (Accessed on 03/16/2020). 3
- [GPMSS*18] GARCIA PINTO V., MELLO SCHNORR L., STANISIC L., LEGRAND A., THIBAUT S., DANJEAN V.: A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *Concurrency and Computation: Practice and Experience* 30, 18 (2018), e4472. doi:10.1002/cpe.4472. 3
- [Gre11] GREEN D. A.: A colour scheme for the display of astronomical intensity images. *arXiv preprint arXiv:1108.5083* (2011). URL: <https://arxiv.org/abs/1108.5083>. 7
- [Gre16] GREGG B.: The Flame Graph. *Communications of the ACM* 59, 6 (2016), 48–57. doi:10.1145/2909476. 3
- [GUK17] GONG X., UBAL R., KAELI D.: Multi2Sim Kepler: A detailed architectural gpu simulator. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2017), pp. 269–278. doi:10.1109/ISPASS.2017.7975298. 2, 3
- [IBJ*14] ISAACS K. E., BREMER P.-T., JUSUFI I., GAMBLIN T., BHATELE A., SCHULZ M., HAMANN B.: Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2349–2358. doi:10.1109/TVCG.2014.2346456. 2, 3
- [IBL*15] ISAACS K. E., BHATELE A., LIFFLANDER J., BÖHME D., GAMBLIN T., SCHULZ M., HAMANN B.: Recovering logical structure from Charm++ event traces. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), pp. 1–12. doi:10.1145/2807591.2807634. 3
- [ILG*12] ISAACS K. E., LANDGE A. G., GAMBLIN T., BREMER P.-T., PASCUCCI V., HAMANN B.: Exploring performance data with Boxfish. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (2012), pp. 1380–1381. doi:10.1109/SC.Companion.2012.202. 3
- [KBB*06] KNÜPFER A., BRENDL R., BRUNST H., MIX H., NAGEL W. E.: Introducing the open trace format (OTF). In *International Conference on Computational Science* (2006), pp. 526–533. doi:10.1007/11758525_71. 5
- [KBD*08] KNÜPFER A., BRUNST H., DOLESCHAL J., JURENZ M., LIEBER M., MICKLER H., MÜLLER M. S., NAGEL W. E.: The vampir performance analysis tool-set. In *Tools for high performance computing* (2008), pp. 139–155. doi:10.1007/978-3-540-68564-7_9. 5
- [KCa] KCachegrind. <http://kcachegrind.sourceforge.net/html/Home.html>. (Accessed on 03/16/2020). 3
- [KL83] KRUSKAL J. B., LANDWEHR J. M.: Icicle plots: Better displays for hierarchical clustering. *The American Statistician* 37, 2 (1983), 162–168. doi:10.2307/2685881. 3
- [KSAR20] KHAIRY M., SHEN Z., AAMODT T. M., ROGERS T. G.: Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), pp. 473–486. doi:10.1109/ISCA45697.2020.00047. 2, 3
- [Lew82] LEWIS C.: *Using the "Thinking-Aloud" Method in Cognitive Interface Design*. IBM TJ Watson Research Center Yorktown Heights, NY, 1982. 8
- [LSJ*19] LI C., SUN Y., JIN L., XU L., CAO Z., FAN P., KAELI D. R., MA S., GUO Y., YANG J.: Priority-based PCIe scheduling for multi-tenant multi-GPU system. *IEEE Computer Architecture Letters* (2019). doi:10.1109/LCA.2019.2955119. 8
- [LTM18] LAM H., TORY M., MUNZNER T.: Bridging from goals to tasks with design study analysis reports. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 435–445. doi:10.1109/TVCG.2017.2744319. 5

- [ME09] McDONNELL B., ELMQVIST N.: Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics* 15 (11 2009), 1105–12. doi:10.1109/TVCG.2009.191. 2
- [MGF11] MATEJKA J., GROSSMAN T., FITZMAURICE G.: Ambient help. In *Proc. SIGCHI Conference on Human Factors in Computing Systems* (2011), CHI '11, p. 2751–2760. doi:10.1145/1978942.1979349. 10
- [MHHH15] MORITZ D., HALPERIN D., HOWE B., HEER J.: Perfoption: Visual query analysis for distributed databases. In *Computer Graphics Forum* (2015), vol. 34, pp. 71–80. doi:10.1111/cgf.12619. 3
- [MM18] MORISHIMA S., MATSUTANI H.: Accelerating blockchain search of full nodes using GPUs. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (2018), pp. 244–248. doi:10.1109/PDP2018.2018.00041. 2
- [MSD*20a] MOJUMDER S. A., SUN Y., DELSHADTEHRANI L., MA Y., BARUAH T., ABELLÁN J., KIM J., KAELI D., JOSHI A., ET AL.: Mgpu-tsm: A multi-gpu system with truly shared memory. *arXiv preprint arXiv:2008.02300* (2020). 8
- [MSD*20b] MOJUMDER S. A., SUN Y., DELSHADTEHRANI L., MA Y., BARUAH T., ABELLÁN J. L., KIM J., KAELI D., JOSHI A.: Halcone: A hardware-level timestamp-based cache coherence scheme for multi-gpu systems. *arXiv preprint arXiv:2007.04292* (2020). 8
- [Mun08] MUNZNER T.: Process and pitfalls in writing information visualization research papers. In *Information visualization*. 2008, pp. 134–153. doi:10.1007/978-3-540-70956-5_6. 10
- [Mun14] MUNZNER T.: *Visualization Analysis and Design*. CRC press, 2014. 4
- [NHKM14] NAVARRO C. A., HITSCHFELD-KAHLER N., MATEU L.: A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics* 15, 2 (2014), 285–329. doi:10.4208/cicp.110113.010813a. 2
- [nvp20] NVIDIA Visual Profiler, 2020. URL: <https://developer.nvidia.com/nvidia-visual-profiler>. 9
- [OJ04] OH K.-S., JUNG K.: GPU implementation of neural networks. *Pattern Recognition* 37, 6 (2004), 1311–1314. doi:10.1016/j.patcog.2004.01.013. 2
- [PDL*11] PONGNUMKUL S., DONTCHEVA M., LI W., WANG J., BOURDEV L., AVIDAN S., COHEN M. F.: Pause-and-Play: Automatically linking screencast video tutorials with applications. In *Proc. 24th Annual ACM Symposium on User Interface Software and Technology* (2011), UIST '11, p. 135–144. doi:10.1145/2047196.2047213. 10
- [Per] Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page. (Accessed on 03/05/2020). 3
- [PGN*19] PENTECOST L., GUPTA U., NGAN E., BEYER J., WEI G.-Y., BROOKS D., BEHRISCH M.: CHAMPVis: Comparative hierarchical analysis of microarchitectural performance. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)* (2019), pp. 55–61. doi:10.1109/ProTools49597.2019.00013. 3
- [ppr] pprof/readme.md at master · google/pprof. <https://github.com/google/pprof/blob/master/doc/README.md>. (Accessed on 03/16/2020). 3
- [rad20] Radeon GPU Profiler, 2020. URL: <https://gpuopen.com/gaming-product/radeon-gpu-profiler-rgp/>. 3
- [Rei05] REINDERS J.: VTune performance analyzer essentials. *Intel Press* (2005). 9
- [Rog11] ROGERS Y.: Interaction design: beyond human-computer interaction, 3rd edition, 2011. 6
- [Ros13] ROSEN P.: A visual approach to investigating shared and global memory behavior of CUDA kernels. In *Computer Graphics Forum* (2013), vol. 32, pp. 161–170. doi:10.1111/cgf.12103. 3
- [RWP*19] ROSS C. J., WOLFE N., PLAGGE M., CAROTHERS C. D., MUBARAK M., ROSS R. B.: Using scientific visualization techniques to visualize parallel network simulations. In *Proc. 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2019), pp. 197–200. doi:10.1145/3316480.3322888. 2, 3
- [RZ04] ROBERTS J. E., ZILLES C.: *TraceVis: an Execution Trace Visualization Tool*. PhD thesis, University of Illinois at Urbana-Champaign, 2004. 3, 6
- [SASK19] SUN Y., AGOSTINI N. B., SHI D., KAELI D.: Summarizing CPU and GPU design trends with product data. *arXiv preprint arXiv:1911.11313* (2019). URL: <https://arxiv.org/abs/1911.11313>. 2
- [SBM*19] SUN Y., BARUAH T., MOJUMDER S. A., DONG S., GONG X., TREADWAY S., BAO Y., HANCE S., MCCARDWELL C., ZHAO V., ET AL.: MGPUSim: Enabling multi-GPU performance modeling and optimization. In *46th International Symposium on Computer Architecture* (2019). doi:10.1145/3307650.3322230. 2, 3, 7
- [SGZ*16] SUN Y., GONG X., ZIABARI A. K., YU L., LI X., MUKHERJEE S., MCCARDWELL C., VILEGAS A., KAELI D.: Hetero-Mark, a benchmark suite for CPU-GPU collaborative computing. In *IEEE International Symposium on Workload Characterization* (2016). doi:10.1109/IISWC.2016.7581262. 8
- [Tho06] THOMAS D. R.: A general inductive approach for analyzing qualitative evaluation data. *American Journal of Evaluation* 27, 2 (2006), 237–246. doi:10.1177/1098214005283748. 4
- [TLCC17] THUDT A., LEE B., CHOE E. K., CARPENDALE S.: Expanding research methods for a realistic understanding of personal visualization. *IEEE Computer Graphics and Applications* 37, 2 (2017), 12–18. doi:10.1109/MCG.2017.23. 10
- [TSAK19] TAVANA M. K., SUN Y., AGOSTINI N. B., KAELI D.: Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-GPU systems. In *33rd International Parallel and Distributed Processing Symposium* (2019). doi:10.1109/IPDPS.2019.00075. 8
- [UJM*12] UBAL R., JANG B., MISTRY P., SCHAA D., KAELI D.: Multi2Sim: a simulation framework for CPU-GPU computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2012), pp. 335–344. doi:10.1145/2370816.2370865. 2, 3, 4
- [Val] Valgrind. <https://valgrind.org/docs/manual/cl-manual.html>. (Accessed on 03/16/2020). 3
- [Ver] Vervysleepy/verysleepy: Very Sleepy, a polling CPU profiler. <https://github.com/VerySleepy/verysleepy>. (Accessed on 03/05/2020). 3
- [XXM18] XIE C., XU W., MUELLER K.: A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2018), 215–224. doi:10.1109/TVCG.2018.2865026. 2, 3
- [YTZ*08] YAN G., TIAN J., ZHU S., DAI Y., QIN C.: Fast cone-beam CT image reconstruction using GPU hardware. *Journal of X-ray Science and Technology* 16, 4 (2008), 225–234. 2
- [ZCD19] ZHANG Y., CHANANA K., DUNNE C.: IDMMVis: Temporal event sequence visualization for type 1 diabetes treatment decision support. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (Jan 2019), 512–522. doi:10.1109/TVCG.2018.2865076. 4, 5
- [ZDBS*19] ZHANG Y., DI BARTOLOMEO S., SHENG F., JIMISON H., DUNNE C.: Evaluating alignment approaches in superimposed time-series and temporal event-sequence visualizations. In *2019 IEEE Visualization Conference (VIS)* (2019), pp. 1–5. doi:10.1109/VISUAL.2019.8933584. 6
- [ZUSK15] ZIABARI A. K., UBAL R., SCHAA D., KAELI D.: Visualization of OpenCL application execution on CPU-GPU systems. In *Proc. Workshop on Computer Architecture Education* (2015), WCAE '15. doi:10.1145/2795122.2795125. 2, 4